Università degli Studi di Bologna
Facoltà di Ingegneria

Progettazione di Applicazioni Web T

Esercitazione 6
Il Pattern DAO: un esercizio completo

Agenda

- **Esercizio guidato completo per farci trovare pronti alla prova d'esame**

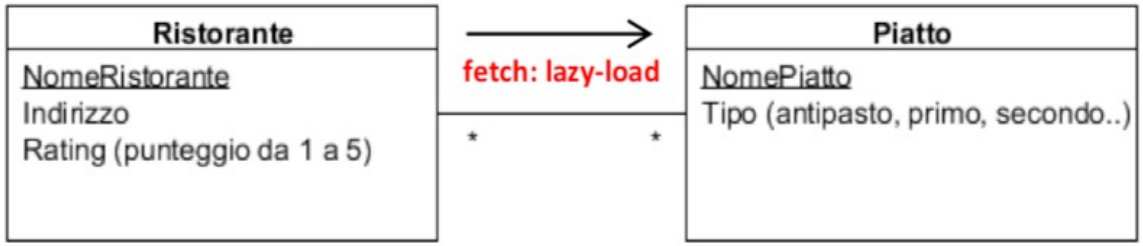
  Progettazione, implementazione e gestione della persistenza basata su metodologia Pattern DAO a partire da una realtà di interesse descritta mediante uno schema UML

- Passi principali:
  - Dall'UML ai JavaBean, dall'UML alle tabelle DB mediate progettazione logica
  - La Relazione M-N e il verso di percorrenza
  - Il fetch Lazy-Load
  - Implementazione delle query

“Offerta Piatti Ristorante”

N.B. Con un piccolo abuso di notazione, nei diagrammi UML la sottolineatura di un attributo non indica la staticità di tale attributo bensì il suo vincolo di univocita alla E/R.

Partendo dalla realtà illustrata nel **diagramma UML** di seguito riportato, si fornisca una soluzione alla gestione della persistenza basata su **Pattern DAO** in grado di “mappare” efficientemente e con uso di ID surrogati il modello di dominio rappresentato dai **JavaBean Ristorante e Piatto del diagramma UML** con le corrispondenti **tabelle relazionali derivate dalla progettazione logica del diagramma** stesso.



Nel dettaglio, dopo aver creato da applicazione Java gli **schemi delle tabelle** all’interno del proprio schema nel database **TW_STUD** di **DB2** (esplicitando tutti i **vincoli** opportuni), **implementato** i **JavaBean** e **realizzato le classi** relative al **Pattern DAO** per **l’accesso CRUD** alle tabelle, si richiede l’implementazione di **opportuni metodi per il supporto delle seguenti operazioni**:

- per ogni ristorante sito in Bologna, si richiede la lista dei piatti di tipo “primo” offerti nel rispettivo menù; si richiede quanti ristoranti, nella fascia di rating compresa tra 4 e 5, offrono come tipo di secondo piatto “seppie con i piselli”.

Si crei poi un **main di prova** che: (i) inserisca due o piu’ tuple nelle tabelle di interesse; (ii) faccia uso corretto dei metodi realizzati al punto precedente al fine di produrre una stampa del risultato sul file **ristorante.txt**.

Primo approccio all'esercizio

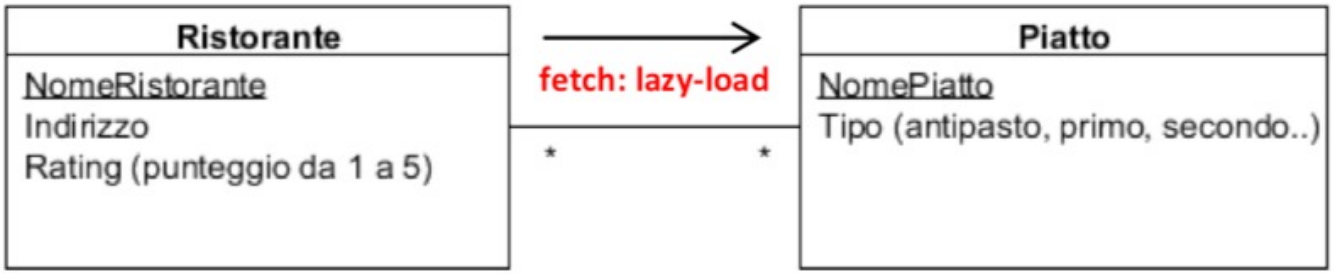
Punti salienti:

- L'utilizzo del **Pattern DAO**
- La relazione **M-N** e il suo **verso di percorrenza**
- La tipologia di fetching: **Lazy-Load**

Note: abbiamo già visto cosa vuol dire utilizzare gli ID surrogati, la scrittura su file di testo dei risultati la diamo per assodata

Dall'UML al Patter DAO e alle tabelle DB

- La prima cosa da fare è trasformare il diagramma UML nelle corrispondenti classi Java e tabelle DB (derivate dalla progettazione logica applicata all'UML)



Dalla teoria sappiamo che il Pattern DAO ha due tipologie di entità Java (Classi ed Interfacce) principali:

- **Data Transfer Object (DTO):** sono gli oggetti che incapsulano i dati veri e propri
- **Data Access Object:** sono interfacce ed oggetti che mappano le operazioni relative alla persistenza sugli oggetti DTO

Dall'UML al Patter DAO

- Come detto gli oggetti DTO incapsulano i dati veri e propri, nel nostro esercizio quindi avremo:
  - **RistoranteDTO**
  - **PiattoDTO**
- A livello implementativo le classi Java conterranno tutti gli attributi indicati nell'UML, in più però, RistoranteDTO (solamente) conterrà una Collezione di PiattoDTO

- Per le entità DAO bisognerà fare un'interfaccia ed una classe Java di implementazione (almeno ;-)) per ogni classe del diagramma UML, quindi nel nostro caso:
  - Le interfacce: **RistoranteDAO** e **PiattoDAO**
  - Le implementazioni: **Db2RistoranteDAO** e **Db2PiattoDAO**

Dall'UML al Patter DAO: un po' di codice

```java
public class RistoranteDTO implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private int id;
    private String nomeRistorante;
    private String indirizzo;
    private int rating;
    private List<PiattoDTO> piatti;
```

```java
public class PiattoDTO {

    private String nomePiatto;
    private String tipo;
    private int id;
```

```java
public interface RistoranteDAO {

    // --- CRUD ------------

        public void create(RistoranteDTO ristorante);

        public RistoranteDTO read(String nome);

        public boolean update(RistoranteDTO ristorante);

        public boolean delete(String nome);


        // --------------------------------
        public List<RistoranteDTO> getResturantByCity(String citta);

        public List<RistoranteDTO> getRatedResturant(int stars);
        // --------------------------------

        public boolean createTable();

        public boolean dropTable();

}
```

```java
public interface PiattoDAO {
    // --- CRUD ------------

    public void create(PiattoDTO piatto);

    public PiattoDTO read(String nome);

    public boolean update(PiattoDTO piatto);

    public boolean delete(String nome);


    // --------------------------------



    public boolean createTable();

    public boolean dropTable();
}
```

La relazione M-N e il verso di percorrenza

Abbiamo già visto che una relazione UML molti a molti si mappa con una classe extra nel mondo relazionale Java, detta classe di mapping.

Nel Pattern DAO una relazione molti a molti si mappa alla stessa maniera, aggiungendo un'interfaccia DAO e la sua implementazione

Nel nostro caso:

- RistorantePiattoMappingDAO
- Db2RistorantePiattoMappingDAO

Il verso di percorrenza (la freccia sopra la relazione) indica la navigabilità della relazione, ovvero il verso di "lettura" dei dati. Nel nostro caso da Ristorante verso Piatto:

Ecco perchè solo RistoranteDTO ha una Collezione di PiattoDTO!!

La relazione M-N e il verso di percorrenza: nella pratica

```java
public interface RistorantePiattoMappingDAO {

    // --- CRUD -------------

        public void create(int idr, int idp);

        //public RistorantePiattoMappingDTO read(int idRistorante, int idPiatto);

        //public boolean update(CourseDTO student);

        public boolean delete(int idRistorante, int idPiatto);


        // ---------------------------------

        public List<PiattoDTO> getPiattiFromResturant(int id);


        // ---------------------------------


        public boolean createTable();

        public boolean dropTable();
}
```

```java
public class Db2RistorantePiattoMappingDAO implements
        RistorantePiattoMappingDAO {

    // === Costanti letterali per non sbagliarsi a scrivere !!! ==========
        static final String TABLE = "ristoranti_piatti";
        // ------------------------------------------------------------
        static final String ID_R = "idRistorante";
        static final String ID_P = "idPiatto";

        // == STATEMENT SQL ==========================================

        // INSERT INTO table ( idCourse, idStudent ) VALUES ( ?,? );
        static final String insert = ;

        // SELECT * FROM table WHERE idcolumns = ?;
        static String read_by_ids = ;

        // SELECT * FROM table WHERE idcolumns = ?;
        static String read_by_ristoranteID = ;

        // SELECT * FROM table WHERE idcolumns = ?;
        static String read_by_piattoID = ;

        // SELECT * FROM table WHERE idcolumns = ?;
        static String dish_query = ;

        // SELECT * FROM table WHERE stringcolumn = ?;
        static String read_all = ;

        // DELETE FROM table WHERE idcolumn = ?;
        static String delete = ;

        // UPDATE table SET xxxcolumn = ?, ... WHERE idcolumn = ?;
        /*static String update = ;

        // SELECT * FROM table;
        static String query = ;

        // CREATE entrytable ( code INT NOT NULL PRIMARY KEY, ... );
        static String create =
            "CREATE " +
                "TABLE " + TABLE +" ( " +
                    ID_R + " INT NOT NULL, " +
                    ID_P + " INT NOT NULL, " +
                    "PRIMARY KEY (" + ID_R +","+ ID_P + " ), " +
                    "FOREIGN KEY ("+ID_R+") REFERENCES ristoranti(id), "+
                    "FOREIGN KEY ("+ID_P+") REFERENCES piatti(id) "+
                ") "
            ;

        static String drop = ;
```

Il Fetch Lazy-Load

- Come sappiamo dalla teoria, la tipologia di fetching Lazy-Load prevede il caricamento dei dati di tipo “pigro”, ovvero questi vengono caricati solo quando sono strettamente necessari, ovvero immediatamente prima che siano processati.

- Come si implementa il fetching Lazy-Load in Java, e come si integra nel Pattern DAO?

  Serve una classe PROXY, ovvero occorre:

  Estendere RistoranteDTO creando una classe proxy

  Db2RistoranteDTOProxy

  Questa classe farà l’override della getPiatti() della superclasse andando a caricare dal db tutti i piatti che serve lo specifico oggetto RistoranteDTO

  La query per il caricamento dei piatti del ristorante sarà contenuta nell’interfaccia RIstorantePiattoMappingDAO

Il Fetch Lazy-Load: codice (1)

La classe proxy Db2RistoranteDTOProxy

```
public class Db2RistoranteDTOProxy extends RistoranteDTO {

    public Db2RistoranteDTOProxy() {
        super();
        // TODO Auto-generated constructor stub
    }


    @Override
    public List<PiattoDTO> getPiatti()
    {
        if(isAlreadyLoaded())
            return super.getPiatti();
        else
        {
            RistorantePiattoMappingDAO rpm = new Db2RistorantePiattoMappingDAO();
            isAlreadyLoaded(true);
            return rpm.getPiattiFromResturant(this.getId());
        }
    }

}
```

Ad ogni chiamata del metodo getPiatti(), la classe controlla che i dati non siano già stati caricati precedentemente, evitando, in tal caso, di andare un'altra volta sul DB

Il Fetch Lazy-Load: codice (2)

Il lazy-load lato DB

```
public interface RistorantePiattoMappingDAO {

    // --- CRUD ------------

        public void create(int idr, int idp);

        //public RistorantePiattoMappingDTO read(int idRistorante, int idPiatto);

        //public boolean update(CourseDTO student);

        public boolean delete(int idRistorante, int idPiatto);


        // --------------------------------

        public List<PiattoDTO> getPiattiFromResturant(int id);

        // --------------------------------


        public boolean createTable();

        public boolean dropTable();
}
```

```
static String dish_query =
    "SELECT * " +
        "FROM " + TABLE + " RP, piatti P " +
        "WHERE RP.idPiatto = P.id AND " + ID_R + " = ? ";
```

```
@Override
public List<PiattoDTO> getPiattiFromResturant(int id) {
    List<PiattoDTO> result = null;
    if ( id< 0 )  {
        System.out.println("read(): cannot read an entry with a negative id");
        return result;
    }
    Connection conn = Db2DAOFactory.createConnection();
    try {
        PreparedStatement prep_stmt = conn.prepareStatement(dish_query);
        prep_stmt.clearParameters();
        prep_stmt.setInt(1, id);
        ResultSet rs = prep_stmt.executeQuery();

        result = new ArrayList<PiattoDTO>();
        while ( rs.next() ) {
            PiattoDTO entry = new PiattoDTO();
            entry.setId((rs.getInt("id")));
            entry.setNomePiatto(rs.getString("nome"));
            entry.setTipo(rs.getString("tipo"));
            result.add(entry);
        }
        rs.close();
        prep_stmt.close();
    }
    catch (Exception e) {

        e.printStackTrace();
    }
    finally {
        Db2DAOFactory.closeConnection(conn);
    }
    return result;
}
```

Le Query

Una volta progettata bene tutta l'architettura relazionale in Java, le query non risultano di particolare difficoltà:

- Restituire tutti i primi piatti dei ristoranti siti in Bologna
- Contare i ristoranti con voti tra 4 e 5 (compresi) che servono "Seppie e Piselli"

La Prima Query: Primi Piatti a Bologna (1)

- Metodo nella classe di Test: riceve tutti i ristoranti di Bologna e ne prende i primi piatti

```
public static String ListPrimiPiattiDeiRistorantiDiBologna(RistoranteDAO r, RistorantePiattoMappingDAO rpm)
{
    List<RistoranteDTO> ristorantiBolognesi = r.getResturantByCity("Bologna");
    String result="";
    for(RistoranteDTO risto : ristorantiBolognesi)
    {
        boolean trovato=false;
        List<PiattoDTO> primiPiatti = risto.getPiatti();
        for(PiattoDTO p : primiPiatti)
        {
            if(p.getTipo().compareTo("primo")==0)
            {

                result = result+p+"\n";
                trovato=true;
            }
        }
        if(trovato)
        {

            result = result+"Questo/i piatto/i è/sono preparato/i da: "+risto.getNomeRistorante()+"\n";
        }
    }
    return result;
}
```

La Prima Query: Primi Piatti a Bologna (2)

Lato DB,
classe
Db2RistoranteDAO

```java
@Override
public List<RistoranteDTO> getResturantByCity(String citta) {
    // TODO Auto-generated method stub
    List<RistoranteDTO> result = null;
    // --- 2. Controlli preliminari sui dati in ingresso ---
    if ( citta.isEmpty() || citta == null )  {
        return result;
    }
    // --- 3. Apertura della connessione ---
    Connection conn = Db2DAOFactory.createConnection();
    // --- 4. Tentativo di accesso al db e impostazione del risultato ---
    try {
        // --- a. Crea (se senza parametri) o prepara (se con parametri) lo statement
        PreparedStatement prep_stmt = conn.prepareStatement(query);
        // --- b. Pulisci e imposta i parametri (se ve ne sono)
        // --- c. Esegui l'azione sul database ed estrai il risultato (se atteso)
        ResultSet rs = prep_stmt.executeQuery();
        result = new ArrayList<RistoranteDTO>();
        // --- d. Cicla sul risultato (se presente) per accedere ai valori di ogni sua tupla
        String address;
        while ( rs.next() ) {
            address = rs.getString("indirizzo").toLowerCase();
            if(address.contains("bologna "))
            {
                RistoranteDTO entry = new Db2RistoranteDTOProxy();
                entry.setId(rs.getInt(ID));
                entry.setIndirizzo(address);
                entry.setRating(rs.getInt(RATING));
                entry.setNomeRistorante(rs.getString(NOMERISTORANTE));
                result.add(entry);
            }
        }
        // --- e. Rilascia la struttura dati del risultato
        rs.close();
        // --- f. Rilascia la struttura dati dello statement
        prep_stmt.close();
    }
    // --- 5. Gestione di eventuali eccezioni ---
    catch (Exception e) {
        e.printStackTrace();
    }
    // --- 6. Rilascio, SEMPRE E COMUNQUE, la connessione prima di restituire il controllo al chiamante
    finally {
        Db2DAOFactory.closeConnection(conn);
    }
    // --- 7. Restituzione del risultato (eventualmente di fallimento)
    return result;
}
```

La Seconda Query: Seppie e Piselli gourmet (1)

- Metodo nella classe di Test: riceve tutti i ristoranti votati almeno con 4 stelle, e controlla se servono “Seppie e Piselli”

```
public static String CountRatedResturantsWithSeppieEPiselli(RistoranteDAO r, RistorantePiattoMappingDAO rpm)
{
    List<RistoranteDTO> ristorantiStellati = r.getRatedResturant(4);
    int counter=0;

    for(RistoranteDTO risto : ristorantiStellati)
    {

        List<PiattoDTO> primiPiatti = risto.getPiatti();
        for(PiattoDTO p : primiPiatti)
        {
            if(p.getNomePiatto().compareTo("Seppie e Piselli")==0)
            {
                counter++;
                break;
            }
        }

    }
    return "Sono stati trovati "+counter+" ristoranti con almeno 4 stelle che preparano Seppie e Piselli";

}
```

La Seconda Query: Seppie e Piselli gourmet (2)

Lato DB,

classe

Db2RistoranteDAO

```
static String find_resturant_over_rate =
        read_all +
            "WHERE " + RATING + " > ? ";
```

```
@Override
public List<RistoranteDTO> getRatedResturant(int stars) {
            List<RistoranteDTO> result = null;
            // --- 2. Controlli preliminari sui dati in ingresso ---
            if ( stars<1 || stars >5 )  {

                return result;
            }
            // --- 3. Apertura della connessione ---
            Connection conn = Db2DAOFactory.createConnection();
            // --- 4. Tentativo di accesso al db e impostazione del risultato ---
            try {
                // --- a. Crea (se senza parametri) o prepara (se con parametri) lo statement
                PreparedStatement prep_stmt = conn.prepareStatement(find_resturant_over_rate);
                // --- b. Pulisci e imposta i parametri (se ve ne sono)
                prep_stmt.clearParameters();
                prep_stmt.setInt(1, stars-1);
                // --- c. Esegui l'azione sul database ed estrai il risultato (se atteso)
                ResultSet rs = prep_stmt.executeQuery();
                result = new ArrayList<RistoranteDTO>();
                // --- d. Cicla sul risultato (se presente) per accedere ai valori di ogni sua tupla
                while ( rs.next() ) {
                        RistoranteDTO entry = new Db2RistoranteDTOProxy();
                        entry.setId(rs.getInt(ID));
                        entry.setIndirizzo(rs.getString(INDIRIZZO));
                        entry.setRating(rs.getInt(RATING));
                        entry.setNomeRistorante(rs.getString(NOMERISTORANTE));
                        result.add(entry);
                }
                // --- e. Rilascia la struttura dati del risultato
                rs.close();
                // --- f. Rilascia la struttura dati dello statement
                prep_stmt.close();
            }
            // --- 5. Gestione di eventuali eccezioni ---
            catch (Exception e) {
                e.printStackTrace();
            }
            // --- 6. Rilascio, SEMPRE E COMUNQUE, la connessione prima di restituire il controllo al chiamante
            finally {
                Db2DAOFactory.closeConnection(conn);
            }
            // --- 7. Restituzione del risultato (eventualmente di fallimento)
            return result;
}
```